Scioto: A Framework for Global-View Task Parallelism*

James Dinan[1], Sriram Krishnamoorthy[2], D. Brian Larkins[1],
Jarek Nieplocha[2], P. Sadayappan[1]

[1] Dept. of Computer Science and Engineering
Dept. of Computer Science
The Ohio State University
Columbus, OH 43221
{dinan, krishnsr, larkins, saday}@cse.ohio-state.edu

[2] Pacific Northwest National Laboratory
Richland, WA 99352
j_nieplocha@pnl.gov

Abstract

We introduce Scioto, Shared Collections of Task Objects, a lightweight framework for providing task management on distributed memory machines under one-sided and global-view parallel programming models. Scioto provides locality aware dynamic load balancing and interoperates with MPI, ARMCI, and Global Arrays. Additionally, Scioto's task model and programming interface are compatible with many other existing parallel models including UPC, SHMEM, and CAF. Through task parallelism, the Scioto framework provides a solution for overcoming irregularity, load imbalance, and heterogeneity as well as dynamic mapping of computation onto emerging architectures. In this paper, we present the design and implementation of the Scioto framework and demonstrate its effectiveness on the Unbalanced Tree Search (UTS) benchmark and two quantum chemistry codes: the closed shell Self-Consistent Field (SCF) method and a sparse tensor contraction kernel extracted from a coupled cluster computation. We explore the efficiency and scalability of Scioto through these sample applications and demonstrate that is offers low overhead, achieves good performance on heterogeneous and multicore clusters, and scales to hundreds of processors.

1 Introduction

Task-parallel decomposition is a popular technique often used to express parallelism in programs that exhibit irregular, sparse, or nested parallelism[28]. In such programs, static partitioning of the computation can lead to load imbalance due to irregularity in the problem space, sparsity in the data, or lack of parallel slackness. By decomposing the problem into tasks and dynamically mapping the computation onto available resources, these classes of applications are able to achieve scalable parallel performance.

In addition to overcoming irregularity in the computation, dynamic task scheduling can also be used to mitigate irregularity present in the hardware. Modern trends in parallel computer architecture have given rise to heterogeneity among processors, heterogeneous cores within processors, and increasing nonuniformity in the memory hierarchy. Clusters of multi- and future many- core processors introduce additional opportunities for dynamic mapping of computation to exploit data locality that is present in these mixed shared and distributed memory systems.

Existing parallel programming tools provide developers with the basic mechanisms needed to implement and manage task parallelism, however these primitives are often low level requiring significant effort to achieve scalable performance [10]. Many popular parallel programming models, such as MPI [18], provide the programmer with a fixed, process-centric view of the computation, requiring them to either statically partition the computation, potentially leading to imbalance, or to construct a higher-level system to manage their parallelism.

One-sided parallel programming models such as MPI-2 [19], SHMEM [2], ARMCI [21], and GASNET [7] offer an alternative to conventional message passing by providing the ability to perform asynchronous accesses to data stored on remote nodes, often through hardware supported RDMA operations. Partitioned Global Address Space (PGAS) models such as Global Arrays (GA) [22], UPC [27], and Co-Array Fortran (CAF) [23] build on these communication primitives by providing the programmer with uniform and partitioned global views of physically distributed shared data objects. These programming models also offer an alternative to conventional Distributed Shared Memory (DSM) systems that promises higher parallel performance by allowing the programmer explicit control over data placement, consistency, and communication. The asynchronous data access model these systems support also allows for natural implementation of applications that exhibit irregular access patterns to shared data. However, many one-sided and PGAS programming models do not provide support for the dynamic computation mapping required in order to support these applica-

*This research was supported in part by DOE grant #DE-FC02-06ER25755 and NSF grant #0403342.

tions.

Emerging high productivity parallel programming languages including Chapel [8], X10 [9], and Fortress [25] offer global views of the data and flexible, high level constructs for expressing parallelism. Under these models, the jobs of managing parallelism and data locality are taken on by the compiler and runtime systems. Scalable and efficient runtime systems for these languages on distributed-memory computers is an active area of research and remains a challenging problem.

Scioto aims to address these gaps by providing a scalable framework that supports the dynamic creation, load balancing, and execution of concurrent tasks that operate within one-sided and global address space models. With Scioto, we extend these high-level global view data models with a runtime system that supports a dynamically scheduled, task-parallel view of the computation and achieves scalable performance on distributed memory machines through locality-aware task placement. In this paper, we describe an implementation of Scioto that interoperates with MPI, ARMCI, and Global Arrays. In addition, Scioto's programming interface and task model are agnostic to the specific platform providing support for the global data view and can be used to provide similar services for a variety of one-sided and GAS models. We evaluate the performance of the Scioto framework on three applications: the Unbalanced Tree Search (UTS) benchmark [10, 24], Self-Consistent Field (SCF) computation [26], and a sparse tensor contraction kernel [4].

We begin our discussion with a conceptual overview of the Scioto task model in Section 2, followed by a more in-depth description of the programming interface in Section 3. In Section 4 we give an example Global Arrays program that uses Scioto for task management. We present the implementation of Scioto in Section 5 followed by an experimental evaluation in Section 6.

2 Overview

Our goal when designing the Scioto framework was to create a general and high level framework for lightweight task management and locality aware dynamic load balancing that can be used both directly and as a compilation target. We have striven to design a task model and programming interface that is interoperable with a variety of existing one-sided and global address space models including MPI-1, MPI-2, UPC, CAF, and Global Arrays. In this work we demonstrate this model in the context of the global address space constructed using ARMCI [21]. In addition to ARMCI's global address space, we demonstrate interoperability with MPI and the Global Arrays (GA) Toolkit. GA is parallel programming tool that is built on ARMCI and provides a high level interface to distributed shared multidimensional arrays.

2.1 Exposing Parallelism Though Task Objects

In a Scioto task-parallel application, the programmer exposes parallelism by adding task objects to a shared *task collection*, shown in Figure 2, and then collectively processing the task collection in an MIMD parallel region. In the context of Scioto, *task objects* are independent, transferable units of work. Each task is described by its *task descriptor* which contains task meta-information as well as user-supplied task arguments. As shown in Figure 1, task descriptors are contiguous objects that contain a standard header where task meta-information is stored and a *task body* where user-supplied task arguments are stored. The user views the task body as a contiguous buffer with a configurable size where they can store any arguments they wish to provide to the task in any format. For example, this space can be used to store structs or arrays and may contain portable references to shared data or common local objects.

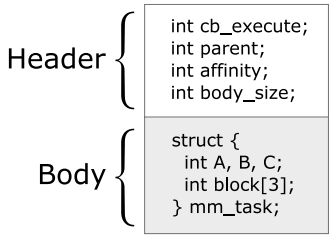

Figure 1. A *task descriptor* is a contiguous object with a header that contains task meta-data and a body that contains task arguments.

When a task is executed, it is provided with a reference to the task collection it is executing on as well as a pointer to its task descriptor. Tasks may gather inputs from their task descriptor, from common local objects, or from shared data in the global space. As a result of its execution, a task may create new subtasks (e.g. continuations) or store results in the shared global space or common local objects. Under the current model, we focus on providing support for independent tasks. *Independent tasks* are tasks that do not rely on the values produced by, or operations performed by other tasks in order to execute. Once started, these tasks must be able to execute to completion when executed in any order and with any degree of concurrency. Tasks may synchronize through locking or other atomic operations when accessing shared data, however a consistent locking discipline must be adopted that ensures progress and avoids circular waiting that can lead to deadlock.

2.2 Remote Access to Shared Data

Under Scioto, tasks are permitted to read from and write to shared data. Many different mechanisms for asynchronous data sharing on distributed clusters are available. One-sided communication libraries such as MPI-2 [19], ARMCI [21], SHMEM [2], or GASNET [7] allow processes to expose regions of their address space for remote access. These tools provide primitives for allocating globally accessible regions of memory, performing remote get and put operations, and synchronizing accesses to shared data. Additionally, some provide wait-free synchronization mechanisms such as atomic swap or atomic increment that allow for high

degrees of concurrency. Higher level languages and tools such as UPC [27], CAF [23], or Global Arrays [22] allow the user to create high level shared data structures such as distributed multidimensional arrays and to interact with distributed global data objects through uniform or partitioned mechanisms.

Scioto's task model is intended to support any of these data sharing mechanisms by allowing the user to place portable references to global data objects in the body of a task. For example, under many one-sided models for communication, a portable shared pointer is the tuple $\langle process, address \rangle$ where $process$ refers to the process that has the data and $address$ is address of the data in $process$'s address space. Some PGAS languages, such as UPC, provide language-level global pointers which can be easily placed within a task body. Under Global Arrays uses, integers are used to provide portable references to arrays and their indices as shown in the example in Figure 1.

2.3 Common Local Objects

In addition to performing input and output with respect to the global space, tasks are also permitted to access common local objects. *Common local objects* are local data objects that are common across all processes. In other words, these are data objects where instances of the same type of object (with possibly differing values) are available in the local address space on all processes in the computation. These objects must be collectively registered with Scioto, which provides a portable reference so that no matter where a task executes, it is able to look up the instance of the object that is local. This functionality can be used to enhance performance by locally gathering intermediate results throughout a computation. It is also key for interoperability with MPI because it provides the only mechanism whereby tasks can produce results due to MPI's lack of global address space.

3 Programming Interface

Scioto presents the programmer with a global view of a distributed collection of tasks referred to as a *task collection*. This task collection is implemented as an aggregation of queues stored on each processor as shown in Figure 2. The set of tasks is distributed across all patches of the collection and each task is assigned an affinity with respect to the local process. When selecting tasks for execution, tasks with the highest affinity are processed first and tasks with lower affinity are given lower priority. When load balancing is performed, low affinity tasks are given the highest priority to be transferred.

Scioto programs begin and end in the SPMD fashion of ARMCI and MPI and collectively enter into a MIMD *task-parallel region* when processing a task collection. A new task collection is first *seeded* either in parallel or sequentially with an initial workload. Once the task collection has been seeded, all processors participating in the task collection collectively call `tc_process()` to enter a task-parallel phase where tasks will be automatically scheduled for parallel execution across all available compute resources. During this stage, tasks may generate subtasks as they are executed. Alternatively, if the programmer would like to rely on their initial task placement scheme, dynamic load balancing can be disabled prior to entering the task parallel region, allowing for a potential reduction in overhead.

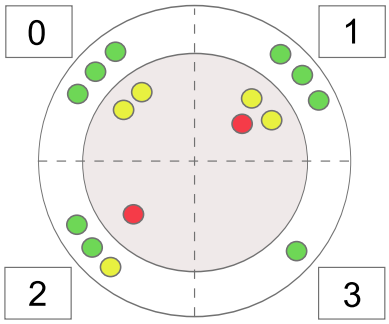

Figure 2. Snapshot of a *task collection* distributed across 4 processors. Tasks are prioritized based on their affinity (color); tasks within the outer ring are private and tasks within the inner disc are shared.

3.1 Scioto Core API

```
tc_t tc_create(int task_sz, int chunk_sz, int max_sz)
void tc_destroy(tc_t tc)
void tc_add(tc_t tc, int proc, int affty, task_t *t)
void tc_process(tc_t tc)
```

A task collection is first created by collectively calling `tc_create()` and providing the maximum size of each task (in bytes), the *chunk size* or granularity for load balancing (in tasks), and the maximum number of tasks that the collection must be able to hold. As discussed in Section 2.1, tasks contain an opaque user-defined body and the task queue allows for multiple different task descriptors to exist in the same queue. However, the user must inform the task collection of the largest task descriptors that they intend to use with each collection (through the `task_sz` parameter) in order to allocate appropriate storage.

Tasks are added to the task collection by calling `tc_add` and providing the process on which to add the task as well as the affinity the task has for the given process. Tasks are added with copy in/out semantics. Thus, when the call to `tc_add()` returns, the `task` buffer is available for reuse. Likewise, tasks are executed on a private copy of the task descriptor allowing this buffer to be reused by the executing task. In situations where tasks are spawned in phases, multiple task collections can be used and processed in sequence. Likewise, once a task collection has been processed, it can be reused by calling `tc_reset()`.

After adding the initial tasks to a collection, it is processed via a collective call to `tc_process()` which enters a MIMD-parallel region where additional tasks may be spawned via calls to `tc_add()`. Only one task collection may be processed at a time, but multiple task collections may

be added to while one is being processed allowing for phase-based task parallelism. The call to `tc_process()` returns collectively when global termination is detected.

3.2 Task Management

```
typedef void (*task_callback_t)(tc_t tc, task_t *t)
int     tc_register_callback(tc_t tc, callback_t fcn)
task_t *tc_task_create(int body_sz, task_handle_t th)
void    tc_task_destroy(task_t *task);
void   *tc_task_body(task_t *task);
```

Every task is represented by its *task descriptor*, shown in Figure 1. A task descriptor wraps a user-defined data structure with a standard header that contains information used by the task collection to schedule and execute the task. From the point-of-view of the Scioto runtime, a descriptor is a `task_t` object with a standard header and an opaque, user-defined body. The header of every task descriptor contains a portable *task callback handle*, `cb_execute`. Task handles are integers that are generated by collective registration of the task's callback function and this handle is used to look up and dispatch the task's callback function when the task is executed.

A task's execution callback function takes a portable reference to the current task collection as well as a local pointer to the descriptor that contains the task's arguments. The task collection reference is passed to each task when it is executed and can be used for creating subtasks or interacting with the runtime system. The API also provides convenient wrappers for managing local task buffers including, creation, destruction, and reuse. The user-defined task body is accessed by calling `tc_task_body()`.

4 Example: Matrix-Matrix Multiplication

Figure 3 provides a code listing for an example Global Arrays program that performs blocked matrix-matrix multiplication on blocked global arrays and the corresponding task descriptor is given in Figure 1. In this example, all processes first collectively create a task collection and register `mm_task_fcn()` as a task callback. This function takes as input a task with an `mm_task` body that contains portable references to the input and output arrays (integers under GA) as well as the indices of the blocks to multiply. Next, a task buffer is created on each process with an `mm_task` body and the `mm_task_fcn()` callback. The body of this task is filled in with references to the global arrays being multiplied. After this, all processes seed the task collection with the multiplication tasks. Each processor creates only the tasks corresponding to patches of the input arrays that are local by calling the user-defined function `get_owner()` and comparing the result with its own process id. After the task has been added, the data in the `task` buffer has been copied into the task collection and the buffer is reused by calling `tc_task_reuse()`.

5 Design and Implementation

Scioto *task collections* are implemented by allocating contiguous arrays of task descriptors in ARMCI shared space on each process. These arrays are treated as a contiguous

```
void mm_task_fcn(tc_t tc, task_t *task) {
  mm_task *mm = tc_task_body(task);
  // Perform multiplication on the given block
}
void main(int argc, char **argv) {
  tc_t     tc;     task_handle_t *hdl;
  task_t  *task;   mm_task       *mm;
  int      A, B, C;

  tc_init(&argc, &argv);
  // Initialize Global Arrays: A, B, and C

  tc   = tc_create(sizeof(mm_task), CHUNK_SIZE,
           MAX_TASKS);
  hdl  = tc_register(tc, mm_task_fcn);
  task = tc_task_create(sizeof(mm_task), hdl);
  mm   = tc_task_body(task);

  mm.A = A; mm.B = B; mm.C = C;

  for (i=0; i < NUM_BLOCKS; i++)
    for (j=0; j < NUM_BLOCKS; j++)
      for (k=0; k < NUM_BLOCKS; k++)
        if (get_owner(i,j,k) == me) {
          mm.block[0] = i;
          mm.block[1] = j;
          mm.block[2] = k;
          tc_add(tc, me, AFFINITY_HIGH, task);
          tc_task_reuse(task);
        }
  tc_process(tc); tc_destroy(tc); tc_finalize();
}
```

Figure 3. Task-parallel blocked matrix-matrix multiplication using Global Arrays.

circular queues and *head* and *tail* indices are maintained to mark the front and back of the queues. Each queue contains a process' current pool of available work and we refer to the collective aggregation of all queues as the *task collection*.

Processes can push and pop tasks to and from both the head and tail of their local queues. Processes are also able to manipulate remote queues using ARMCI one-sided operations. Because queues are contiguous arrays, several tasks can be simultaneously pushed onto or popped off of a remote queue using a single one-sided communication operation.

During an operation on a remote queue, the queue must be locked to prevent updates from colliding. This synchronization leads a reduction in concurrency that can adversely affect the performance of the local process as it can end up waiting for thousands of cycles while a remote process manipulates its queue. In order to reduce contention between local and remote access to shared queues, the queues are *split* into local and shared portions. Local portions of the queue can be accessed by the local process that without locking, while synchronization must be used when accessing the shared portion of the queue. As the computation progresses, processors move tasks between the shared and local portions of the queue to make work available for load balancing or to reclaim shared work for local execution. These operations can be accomplished without copying by simply adjusting the queue's split pointer.

5.1 Dynamic Load Balancing

Scioto uses a locality-awareness enhanced version of work stealing [5]. Under work stealing, processes that have

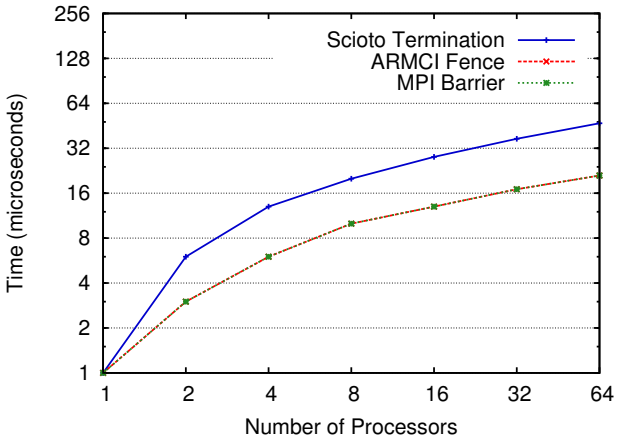

Figure 4. Termination detection, ARMCI Barrier, and MPI Barrier timings on 64 cluster nodes.

exhausted their local work must search among their peers for surplus work. This is done by randomly selecting a peer and performing remote operations on its patch of the task collection to steal surplus work if any is available. In the context of Scioto's split queues, only work that is in the shared region of a process's queue may be stolen. The maximum number of tasks that can be stolen through a single steal operation is referred to as the *chunk size*.

Locality-awareness is implemented by prioritizing the queue such that tasks with high local affinity are placed toward the head of the queue and tasks with low affinity to the local process are placed toward the tail. Steal operations are then performed with respect to the tail of the queue and local task processing is performed with respect to the head of the queue. Thus, tasks with high affinity are most likely to execute on the local process and tasks with low affinity will be the first to be stolen when load balancing is performed.

5.2 Termination Detection

In the context of Scioto, termination occurs when all processes become idle and when no load balancing operations are in progress. In order to detect this, we have implemented a wave-based algorithm similar to that proposed by Francez and Rodeh[11]. In this algorithm, a binary spanning tree is mapped onto the process space and a token wave is passed down and up the tree. The token is initially located at the root of the tree and is split at each node as it is passed down the tree. Once the token wave reaches the leaves, it reverses direction and as nodes become passive and have received their children's tokens, they combine tokens and pass the result up the tree. Tokens are initially colored *white*, however if a node receives a *black* token from one of its children or has performed a load balancing operation since the last down-wave, it must color its token black to signal a re-vote. Token coloring is necessary to ensure that early termination is not detected when a passive process that has already passed a white token up to its parent transitions from passive to active.

Task Collection Operation	Cluster Performance	Cray XT4 Performance
Local Insert	0.4952μs	0.9330μs
Remote Insert	18.0819μs	27.018μs
Local Get	0.3613μs	0.6913μs
Remote Steal	29.0080μs	32.384μs

Table 1. Microbenchmark timings for core Scioto operations.

In the average case, this algorithm requires $log(p)$ messages to detect termination and empirical data comparing the performance of this algorithm with the performance of MPI and ARMCI barriers is presented in Figure 4. In this comparison, we detect termination after executing a single *no-op* task and found that our algorithm can detect termination in roughly twice the time required for ARMCI and MPI barrier operations.

5.3 Token Coloring Optimization

Coloring the token *black* results in additional termination detection waves and, due to one-sided stealing operations, also requires an extra communication operation between the thief and the victim process. This communication is necessary to mark the victim as *dirty*, informing them that they must color their token black to avoid early termination. However, marking the victim as dirty is not necessary under certain conditions, allowing for a reduction in the number of messages.

For the purpose of discussion, we establish a votes-before relation where $p_i \rightarrow p_j$ indicates that p_i casts its vote before p_j. In the case of a binary spanning tree, this means that p_i is a descendant of p_j.

Optimization: The victim, p_v, of a steal operation only needs to be marked as dirty if the thief, p_t, has already voted and $\neg(p_v \rightarrow p_t)$.

Proof: The purpose of p_t marking p_v dirty is for p_t to retract its vote. If $p_v \rightarrow p_t$ then if p_t has voted, p_v must also have already voted and marking p_v dirty will have no effect.

6 Experimental Evaluation

Experiments were conducted on a heterogeneous cluster and a Cray XT4 system. The cluster is configured with 32 2.8GHz AMD Opteron 254 nodes and 32 3.6GHz Intel Xeon nodes. All nodes contain 4GB of memory and 10GBps Mellanox Infiniband NICs. The Cray XT4 is configured with 3,744 2.6Ghz dual-core AMD Opteron 285 Processors running Cray Compute Node Linux. We are presently working on porting Global Arrays to this system and because of this, we are not yet able to report performance results for the SCF and TCE applications on this system.

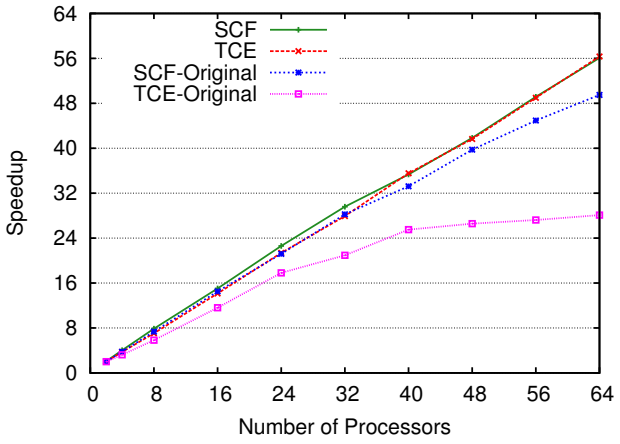

Figure 5. Parallel speedup on 64 cluster nodes for the original and Scioto implementations of SCF and TCE.

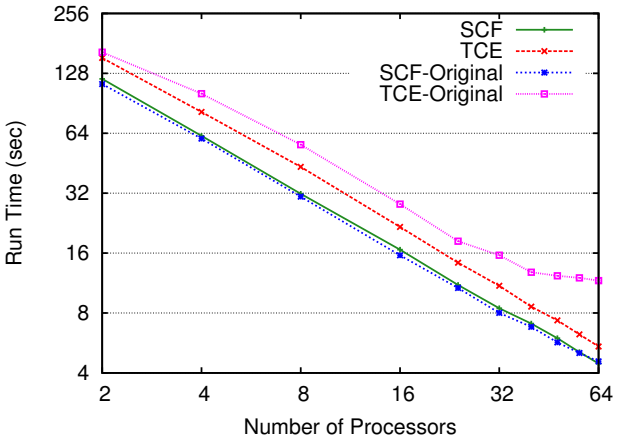

Figure 6. Raw performance of the Original and Scioto implementations of the SCF and TCE applications on 64 cluster nodes.

6.1 Microbenchmark Performance

We constructed several microbenchmarks to measure the performance of core local and remote task collection operations. Results are presented in Table 1 and were collected using a task body size of 1kB and a chunk size of 10. Here we can see that splitting the queue allows for lightweight, lock-free local operations with insert and get taking under $1\mu s$ on both systems. Because the queues have been laid out contiguously for remote access, remote operations also yield good performance with $18\mu s$ and $27\mu s$ average time to add a task to a remote queue and $29\mu s$ and $32\mu s$ average time to perform a steal operation on the cluster and the Cray XT4, respectively.

6.2 Target Applications

We evaluate the performance of Scioto using three applications: the Unbalanced Tree Search (UTS) benchmark, the Self Consistent Field calculation (SCF), and TCE, a representative tensor contraction kernel from the Tensor Contraction Engine. For each application, we have modified the original code to interoperate with Scioto and we compare the performance of the Scioto implementation with the performance of the original version the application:

SCF: We have extended an existing Global Arrays implementation of the closed-shell Self-Consistent Field (SCF) method [26] with Scioto task collections. SCF is a technique commonly used in *ab initio* computational chemistry and involves irregular data access and the accumulation of results into large distributed data structures. This implementation computes the Fock matrix using the non-relativistic SCF method from the Born-Oppenheimer approximation. . Both the Fock and density matrices are distributed across all processors using Global Arrays. In the original implementation, load balancing is achieved by replicating a work queue across all processes and performing atomic increment on a shared counter to get the next available task.

TCE: The TCE application kernel is representative of sparse tensor contraction operations performed by coupled cluster models for *ab initio* electronic structure modeling[4]. In this example, tensor contraction is performed over two block-sparse tensors implemented using Global Arrays and the result is stored into a distributed output global array. This kernel stands to benefit from Scioto due to irregularity introduced though sparsity in the input tensors. The original implementation of TCE also uses a shared global task counter to perform dynamic load balancing.

UTS: The UTS benchmark [10, 24] performs exhaustive parallel search on a deterministic, highly unbalanced search space. The UTS tree traversal starts with a single task and proceeds in nested parallel style to generate millions of tasks, one for each node in the tree. Due to imbalance in the search space and the volume of tasks created, the performance of UTS depends heavily on efficient and lightweight dynamic load balancing. The MPI implementation of UTS dynamically balances the load using a custom work stealing implementation over MPI's two-sided message passing. In our evaluation, we compare the MPI implementation with a modified copy of the same program that uses Scioto for task management. In this implementation, common local objects (see Section 2.3) are used to accumulate the tree statistics gathered by UTS.

6.3 Performance Analysis

In order to show a clear trend on the heterogeneous cluster, we have run our experiments with half Opteron and half Xeon nodes. Thus, doubling the number of nodes also dou-

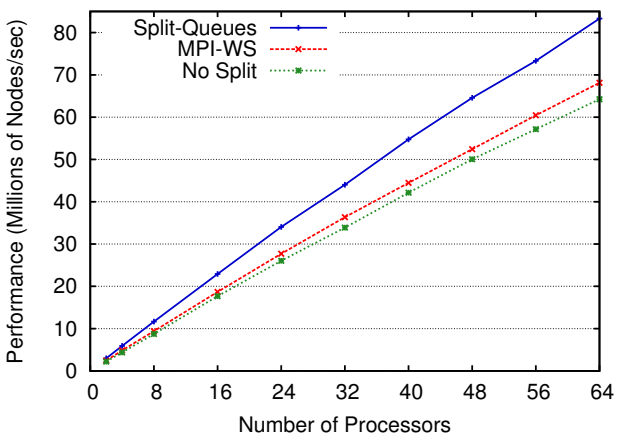

Figure 7. Parallel performance on 64 cluster nodes for the MPI implementation of UTS and the Scioto implementation with and without split queues.

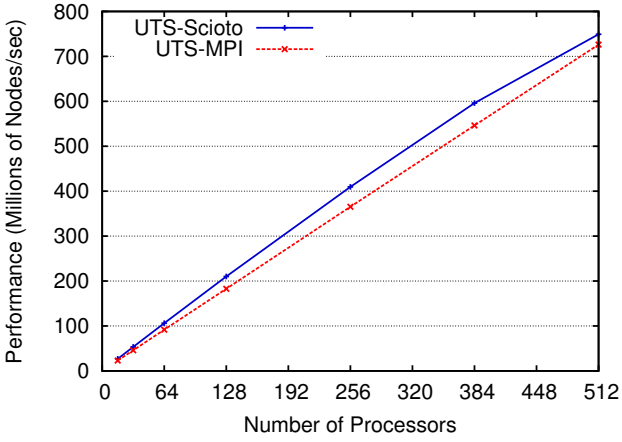

Figure 8. Parallel performance of the UTS benchmark under Scioto and MPI on 512 nodes on the Cray XT4.

bles the resources available to the application even though the processors are not of uniform speed.

Figures 5 and 6 compare the speedup and parallel performance of the original and Scioto implementations of the TCE and SCF applications on the 64 node cluster. TCE shows poor scaling and SCF also shows a slowdown in scaling as we approach 64 processors. This is because both applications rely on a shared global counter to perform locality-oblivious dynamic load balancing. Under this scheme, all processes have the complete list of tasks and the next available task in the list is acquired by atomically incrementing the global counter. In contrast to this, the Scioto implementations of SCF and TCE show better scaling due to Scioto's distributed load balancing scheme. In the case of TCE, the Scioto implementation offers significantly better parallel performance and for SCF, parallel performance is comparable with the global counter scheme up to 32 processors and becomes better as the original implementation of SCF yields lower scaling at 64 processors.

Figures 7 and 8 show the performance in total work units processed per second of the UTS benchmark using Scioto compared with the MPI implementation of the benchmark on the cluster and Cray XT4 systems. Heterogeneity poses a significant challenge to UTS as the performance of the different types of cluster nodes on UTS' SHA-1 implementation varies significantly. On the cluster, Opteron nodes require $0.3158\mu s$ to process a single node on the UTS benchmark and Xeon nodes require $0.4753\mu s$, a 50% difference in performance. In comparison, the Cray XT4 achieves $0.5681\mu s$ per node. In both graphs, we can see that performance is comparable between the Scioto and MPI implementations and that Scioto offers higher performance due to reduced overhead and the elimination of explicit polling operations that must be performed to support work stealing under MPI.

Figure 7 further shows the effectiveness of the split queues compared with our original locked implementation of Scioto. From this data, it is clear that splitting the queue yields a significant increase in concurrency, leading to significantly higher performance than was achievable with both our locked queue implementation and UTS' MPI work stealing load balancer.

7 Related Work

The Scioto projects aims to synthesize a new parallel tasking system with the following properties: 1. High-level support for lightweight task management; 2. Locality-conscious dynamic load balancing; 3. Interoperability with a variety of existing one-sided and global address space programming models; 4. Scalable performance on distributed memory machines.

Several systems have been developed that offer the programmer dynamically scheduled, task-based views of computation. Cilk[12] is a parallel extension to the C programming language that offers an elegant parallel tasking model, allowing the user to fork and join tasks. Cilk's tasking model supports scheduling of fully strict tasks, while, at present, Scioto's task model focuses on independent tasks. Cilk-NOW [6] supports the Cilk tasking model on distributed memory systems and adds fault tolerance, however it focuses primarily on support for functional parallelism and does not support a global address space. Concurrent object systems such as Charm++[14], PREMA[3] or the Illinois Concert System[15] are available on both shared and distributed memory systems and provide adaptive parallel programming models based on concurrently executing migratable objects. These systems provide an object-centric view of the parallel computation rather than a process-centric view and use message passing rather than a global shared address space for data. New high productivity parallel programming languages such as Chapel [8], X10 [9], and Fortress [25] offer language-level primitives to support task parallelism. The

scalable implementation of the powerful task models of these new languages is still a challenging problem and we believe that the implementation and evaluation of the simpler Scioto model on a range of application programs can offer useful insights to their implementors.

Dynamic load balancing and work stealing schemes have been widely investigated [5, 17]. Locality aware load balancing attempts to address communication efficiency by co-locating tasks with their associated data and work has also been done to investigate potential locality benefits from prioritized dynamic load balancing[1].

Distributed Shared Memory systems (DSM) such as Treadmarks [16] and Cluster OpenMP [13] provide the programmer with a global view of distributed shared data. These systems have faced significant performance and scalability challenges on distributed memory systems due to communication overhead. One-sided and PGAS programming models offer high performance by allowing the programmer explicit control over data placement and communication. Multithreaded DSM systems such as DSM-Threads [20] allow the programmer to write dynamically scheduled computation in a global address setting. However, many of these systems rely on fair thread scheduling and thus must multiplex threads to ensure progress and perform full thread migration when balancing the load. Scioto uses a lighter weight approach without fairness guarantees and because of this, does not need to perform task migration.

8 Conclusions

We have presented Scioto, a task parallel framework for one-sided and global address space parallel programming models. Scioto offers an alternative means for expressing parallelism through shared collections of task objects and provides locality-aware dynamic load balancing. We have evaluated the performance of our approach through three applications: SCF, TCE, and UTS and demonstrated that Scioto offers a lightweight and scalable solution for managing parallelism.

We are presently working on extending our independent task model with support for tasks that exhibit arbitrary inter-task dependencies. In our future work, we will also evaluate Scioto with additional global address space parallel models such as UPC. We plan for future enhancements, including wait-free implementations of the distributed task collection, initial placement strategies, and multicore scheduling enhancements.

References

[1] U. A. Acar, G. E. Blelloch, and R. D. Blumofe. The data locality of work stealing. *Theory Comput. Syst.*, 35(3):321–347, 2002.

[2] R. Bariuso and A. Knies. Shmem user's guide, 1994.

[3] K. Barker, A. N. Chernikov, N. Chrisochoides, and K. Pingali. A load balancing framework for adaptive and asynchronous applications. *IEEE Trans. Parallel Distrib. Syst.*, 15(2):183–192, 2004.

[4] G. Baumgartner, A. Auer, D. E. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. J. Harrison, S. Hirata, S. Krishnamoorthy, S. Krishnan, C.-C. Lam, Q. Lu, M. Nooijen, R. M. Pitzer, J. Ramanujam, P. Sadayappan, and A. Sibiryakov. Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models. *Proceedings of the IEEE*, 93(2):276–292, Feb. 2005.

[5] R. Blumofe and C. Leiserson. Scheduling multithreaded computations by work stealing. In *Proc. 35th Ann. Symp. Found. Comp. Sci.*, pages 356–368, Nov. 1994.

[6] R. D. Blumofe and P. A. Lisiecki. Adaptive and reliable parallel computing on networks of workstations. In *ATEC '97: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 10–10, Berkeley, CA, USA, 1997. USENIX Association.

[7] D. Bonachea. GASNet specification, v1.1. Technical Report UCB/CSD-02-1207, U.C. Berkeley, 2002.

[8] B. Chamberlain, D. Callahan, and H. Zima. Parallel programmability and the chapel language. *IJHPCA*, 21(3):291–312, 2007.

[9] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In R. Johnson and R. P. Gabriel, editors, *OOPSLA*, pages 519–538. ACM, 2005.

[10] J. Dinan, S. Olivier, G. Sabin, J. Prins, P. Sadayappan, and C.-W. Tseng. Dynamic load balancing of unbalanced computations using message passing. In *IPDPS*, pages 1–8. IEEE, 2007.

[11] N. Francez and M. Rodeh. Achieving distributed termination without freezing. *Software Engineering, IEEE Transactions on*, SE-8(3):287–292, May 1982.

[12] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proc. 1998 SIGPLAN Conf. Prog. Lang. Design Impl. (PLDI '98)*, pages 212–223, 1998.

[13] Intel Corporation. Cluster OpenMP user's guide v9.1. (309096-002 US), 2005-2006.

[14] L. V. Kalé and S. Krishnan. Charm++: A portable concurrent object oriented system based on c++. In *OOPSLA*, pages 91–108, 1993.

[15] V. Karamcheti and A. A. Chien. Concert-efficient runtime support for concurrent object-oriented programming languages on stock hardware. In *SC*, pages 598–607, 1993.

[16] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proc. of the 18th Annual Int'l Symp. on Computer Architecture (ISCA'92)*, pages 13–21, May 1992.

[17] V. Kumar, A. Y. Grama, and N. R. Vempaty. Scalable load balancing techniques for parallel computers. *J. Par. Dist. Comp.*, 22(1):60–79, 1994.

[18] MPI Forum. MPI: A message-passing interface standard. Technical Report UT-CS-94-230, 1994.

[19] MPI Forum. MPI-2: Extensions to the Message-Passing Interface. Technical Report, University of Tennessee, Knoxville, 1996.

[20] F. Mueller. On the design and implementation of dsm-threads. In H. R. Arabnia, editor, *PDPTA*, pages 315–324. CSREA Press, 1997.

[21] J. Nieplocha and B. Carpenter. ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems. *Lecture Notes in Computer Science*, 1586, 1999.

[22] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global arrays: a portable "shared-memory" programming model for distributed memory computers. In *SC*, pages 340–349, 1994.

[23] R. Numrich and J. Reid. Co-array fortran for parallel programming, 1998.

[24] S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan, and C.-W. Tseng. Uts: An unbalanced tree search benchmark. In G. Almási, C. Cascaval, and P. Wu, editors, *LCPC*, volume 4382 of *Lecture Notes in Computer Science*, pages 235–250. Springer, 2006.

[25] G. L. Steele Jr. Parallel programming and parallel abstractions in fortress. In *IEEE PACT*, page 157. IEEE Computer Society, 2005.

[26] J. L. Tilson, M. Minkoff, A. F. Wagner, R. Shepard, P. Sutton, R. J. Harrison, R. A. Kendall, and A. T. Wong. High performance computational chemistry:(ii) a scalable scf program. In *J. Computational Chemistry*, volume 17, pages 124–132, 1995.

[27] UPC Consortium. UPC language specifications, v1.2. Technical Report LBNL-59208, Lawrence Berkeley National Lab, 2005.

[28] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: Characterization and methodological considerations. In *ISCA*, pages 24–36, 1995.